Números y aritmética

Arquitectura de Computadores – IIC2343

1-2019

Yadran Eterovic

Cuando se trata de números, las personas pensamos en base 10 (números *decimales*):

- diez símbolos diferentes — 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 — llamados dígitos decimales, o simplemente, dígitos

Pero los números pueden ser representados en cualquier base

… p.ej., en base 2 (números *binarios*):

- dos símbolos diferentes — 0, 1 — llamados dígitos binarios o *bits*

Los bits son los “átomos” de la computación:

- toda información se compone de bits

Numeramos los bits de un número binario 0, 1, 2, 3, … de derecha a izquierda:

- es decir, desde el bit menos significativo al bit más significativo

1×10^0

2×10^1

4×10^2

$$\mathbf{4\ 2\ 1}_{10} = \mathbf{1\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 1}_2$$

1×2^0

0×2^1

1×2^2

0×2^3

1×2^8

1×2^7

0×2^6

1×2^5

0×2^4

La cantidad de memoria disponible para almacenar un número queda fija al momento de diseñar el computador:

números de precisión finita

P.ej., el conjunto de enteros positivos representables mediante tres dígitos decimales, sin punto decimal ni signo:

000, 001, 002, …, 999

Es imposible representar ciertos números:

- mayores que 999, negativos, fracciones, irracionales, complejos

El conjunto no es cerrado con respecto a la suma, resta o multiplica-ción:

- 600 + 600 = 1200 → muy grande
- 050 x 050 = 2500 → muy grande

003 – 005 = -2 → negativo

007 / 002 = 3.5 → no es un entero

El álgebra de los números de precisión finita es diferente del álgebra normal:

- p.ej., la ley de asociatividad $a + (b - c) = (a + b) - c$ no se cumple si $a = 700$, $b = 400$ y $c = 300$, porque al calcular $a + b$ en el lado derecho produce *overflow*

Obviamente, no es que los computadores sea especialmente inadecuados para hacer aritmética, sino que

es importante entender cómo funcionan

En computadores en que las palabras tienen 32 bits, podemos representar 2^{32} patrones diferentes de bits

… los números desde el 0 hasta el $2^{32} - 1 = 4{,}294{,}967{,}295_{10}$

Así, con 32 bits, el valor del número representado es

$$\mathbf{(\boldsymbol{x}_{31} \times 2^{31}) + (\boldsymbol{x}_{30} \times 2^{30}) + (\boldsymbol{x}_{29} \times 2^{29}) + \ldots + (\boldsymbol{x}_{1} \times 2^{1}) + (\boldsymbol{x}_{0} \times 2^{0})}$$

Estos números positivos se llaman números sin signo (*unsigned numbers*)

También tenemos que representar números negativos

Podemos agregar un signo, representado en un bit: ***signo y magnitud***

Problemas:

- este bit, ¿va a la derecha o a la izquierda?
- al sumar, se necesita un paso adicional para saber el valor de este bit
- hay un cero positivo y un cero negativo

No habiendo una mejor alternativa obvia

… se eligió la representación que hacía más simple el hardware — ***complemento de 2***:

- 0s a la izquierda significa positivo
- 1s a la izquierda significa negativo

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 0_{10}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = 1_{10}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = 2_{10}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 = 2{,}147{,}483{,}645_{10}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = 2{,}147{,}483{,}646_{10}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = 2{,}147{,}483{,}647_{10}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = -2{,}147{,}483{,}648_{10}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = -2{,}147{,}483{,}647_{10}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = -2{,}147{,}483{,}646_{10}$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 = -3_{10}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = -2_{10}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = -1_{10}$

En 32 bits (diapositiva anterior):

- la mitad positiva, de 0 a $2{,}147{,}483{,}647_{10}$ (= $2^{31} - 1$) usa la misma representación que antes
- el patrón de bits que sigue ($1000...0000_2$) representa el número más negativo, $-2{,}147{,}483{,}648_{10}$ (= -2^{31})

  … el cual no tiene un número positivo correspondiente
- y luego viene una secuencia de números negativos de magnitud decreciente, desde $-2{,}147{,}483{,}647_{10}$ (= $1000...001_2$) hasta -1_{10} (= $1111...1111_2$)

Todo computador hoy en día, y desde 1965, usa complemento de 2 para representar números con signo (*signed numbers*)

string	unsigned	sign magnitude	1's complement	2's complement
0000	0	0	0	**0**
0001	1	1	1	**1**
0010	2	2	2	**2**
0011	3	3	3	**3**
0100	4	4	4	**4**
0101	5	5	5	**5**
0110	6	6	6	**6**
0111	7	7	7	**7**
1000	8	–0	–7	**–8**
1001	9	–1	–6	**–7**
1010	10	–2	–5	**–6**
1011	11	–3	–4	**–5**
1100	12	–4	–3	**–4**
1101	13	–5	–2	**–3**
1110	14	–6	–1	**–2**
1111	15	–7	–0	**–1**

Todos los números negativos tienen un 1 en el bit más significativo:

- *bit de signo*
- basta examinar este bit para saber si un número es positivo o negativo (0 se considera positivo)

Así, con 32 bits, el valor del número representado es

$$(x_{31} \times -2^{31}) + (x_{30} \times 2^{30}) + (x_{29} \times 2^{29}) + \ldots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

Ocurre *overflow* cuando el resultado de la operación produce un bit de signo incorrecto:

- un 0 a la izquierda cuando el número es negativo
- un 1 a la izquierda cuando el número es positivo

Cómo determinamos el inverso aditivo de un número binario Y de n bits en complemento de 2:

- si miramos la diapositiva #12 (o la última columna de la diapositiva #14), vemos que $Y + \bar{Y} = 111...111_2 = -1$
- es decir, $Y + \bar{Y} = -1 \Rightarrow -Y = \bar{Y} + 1$
- es decir, el inverso aditivo $-Y$ se obtiene primero invirtiendo cada 0 de Y a 1 y cada 1 de Y a 0, y luego sumando 1 al resultado
- (por otra parte, si no tomamos en cuenta el signo, $Y + \bar{Y} = 111...111_2 = 2^n$; de aquí el nombre "complemento de 2")

Cómo convertimos un número binario representado en n bits a un número binario representado con más de n bits:

- …

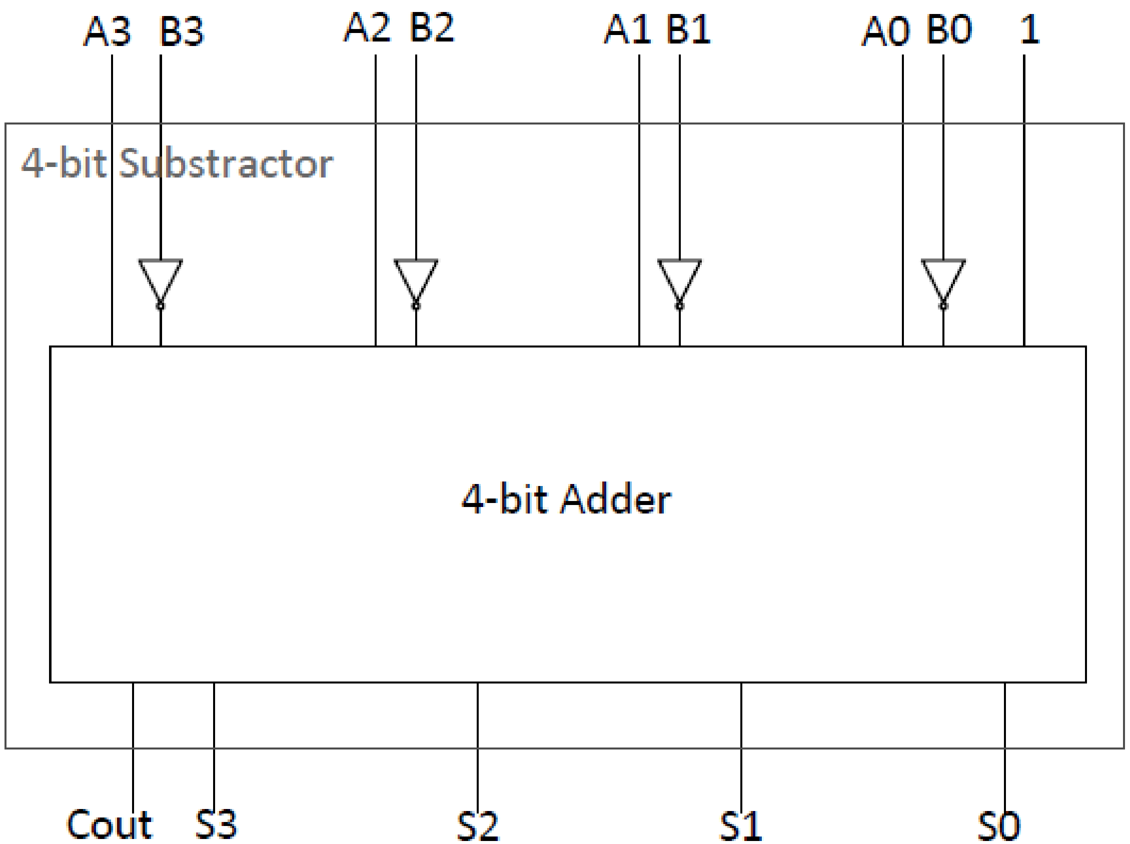
Ahora que podemos representar números negativos y, más en general, el inverso aditivo de un número,

… podemos diseñar un *restador*, a partir del sumador que ya vimos:

- $A - B = A + (-B)$

Es decir, sumamos al minuendo, *A*, el inverso aditivo del sustraendo, *B*:

- primero, invertimos cada bit —B_0, B_1, B_2, B_3— del sustraendo
- … y luego le sumamos 1, … simplemente, poniendo un 1 en el *carry-in*


A3 B3
A2 B2
A1 B1
A0 B0
1
4-bit Substractor
4-bit Adder
Cout
S3
S2
S1
S0

Ahora podemos pensar en construir una unidad sumadora/restadora

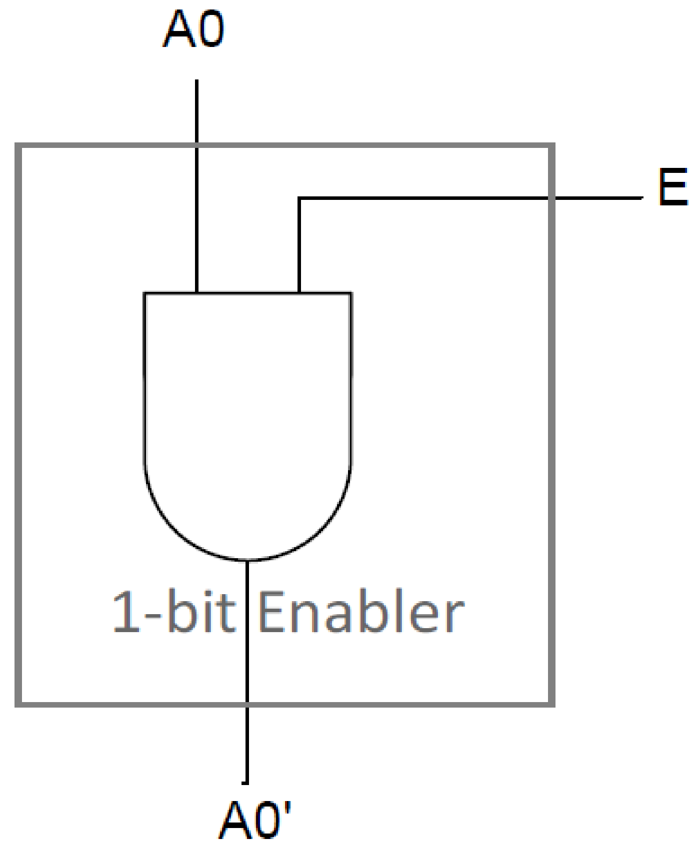
Esta unidad, que consiste en un sumador y un restador, tiene que poder ser controlada, en términos de poder seleccionar la operación que se va a realizar

Así, además de las componentes funcionales —el sumador y el restador—, necesitamos componentes de control:

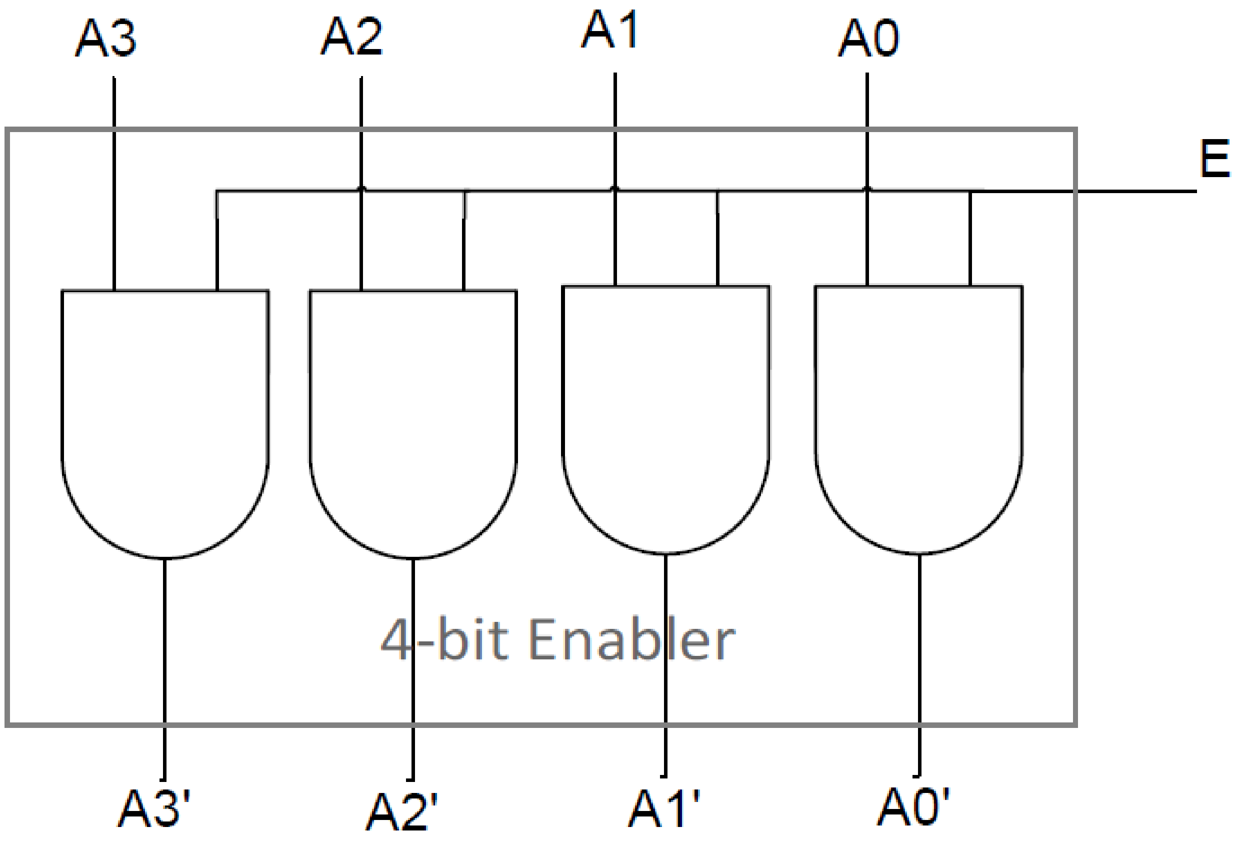
- *enabler* (“habilitador”)
- *multiplexer*, o *mux* (selector)

Enabler de un bit

E	A0'
0	0
1	A0

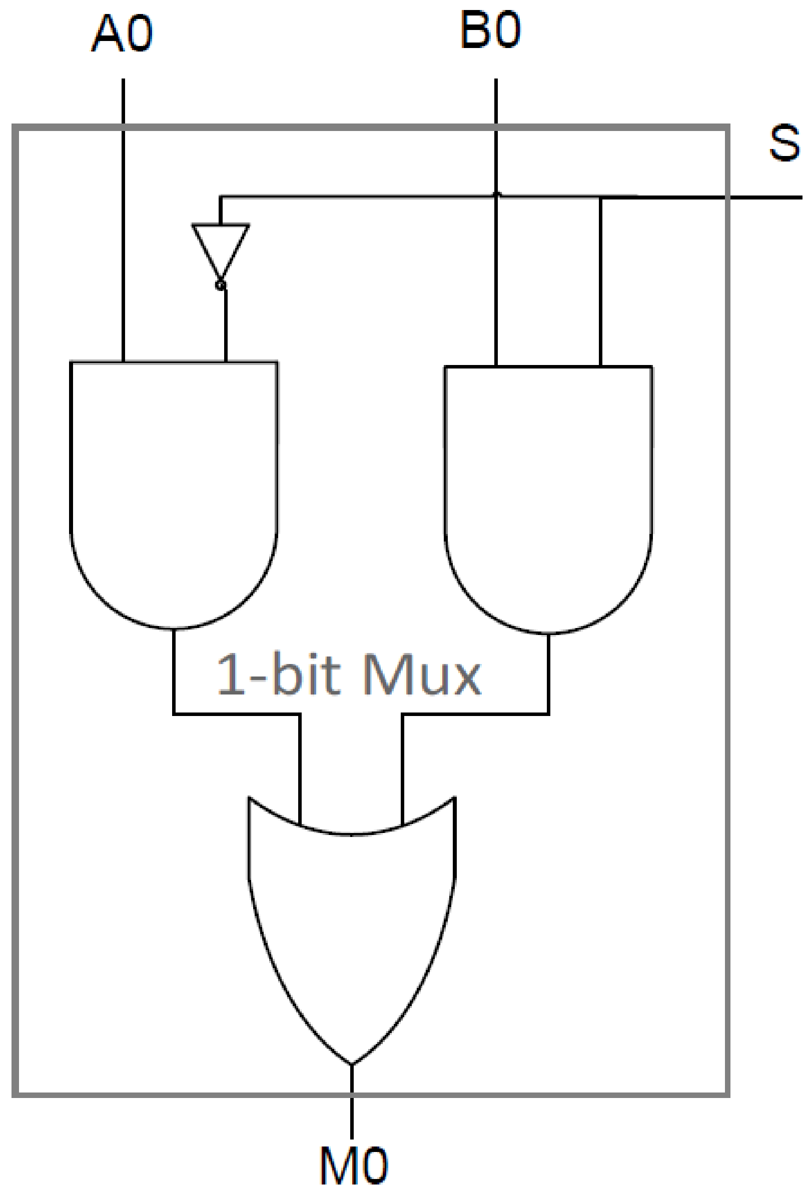


Enabler de cuatro bits

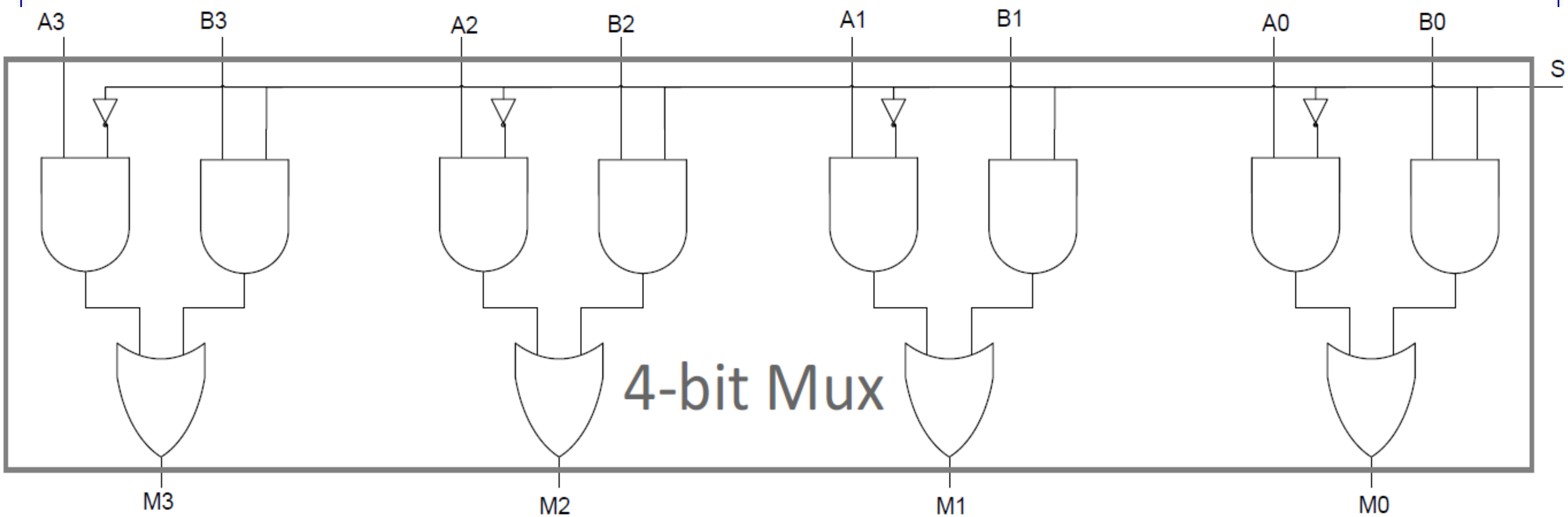


***Multiplexer* de un bit**

S	M0
0	A0
1	B0



Multiplexer de cuatro bits



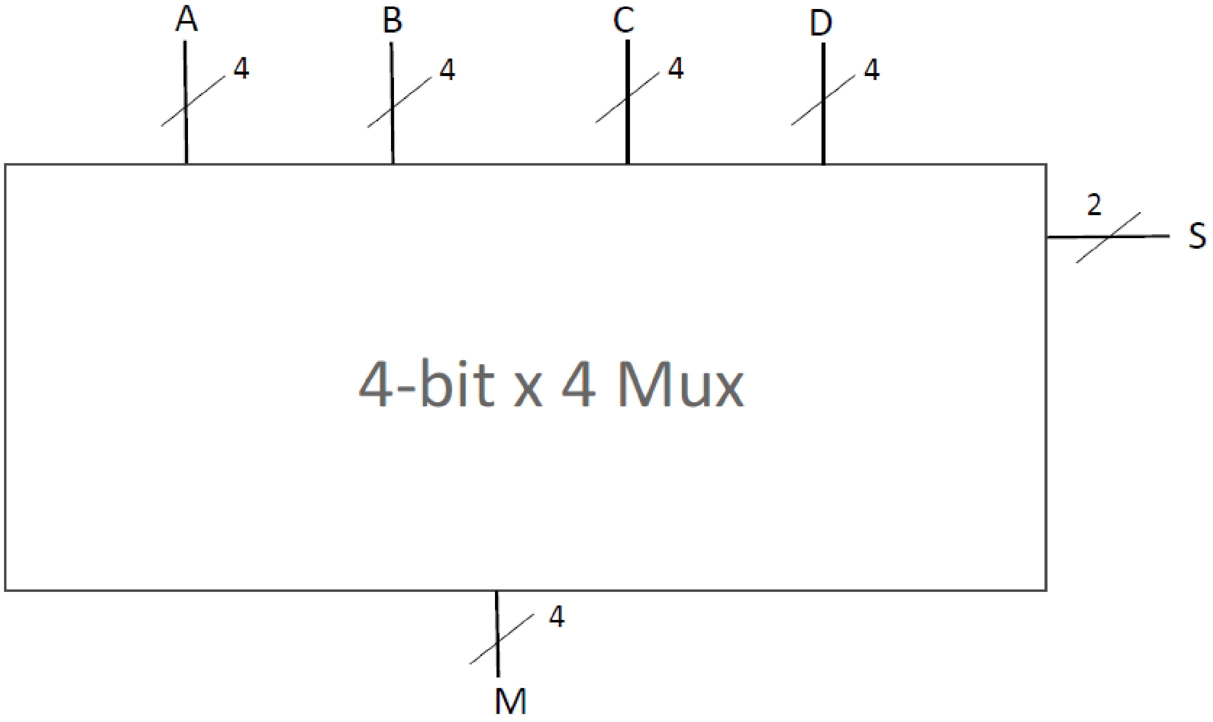
Esto significa que las conexiones (*buses*), que hasta ahora solo transmitían datos (*buses de datos*: los bits que se suman o se restan),

… deben poder transmitir también órdenes de control —*buses de control* :

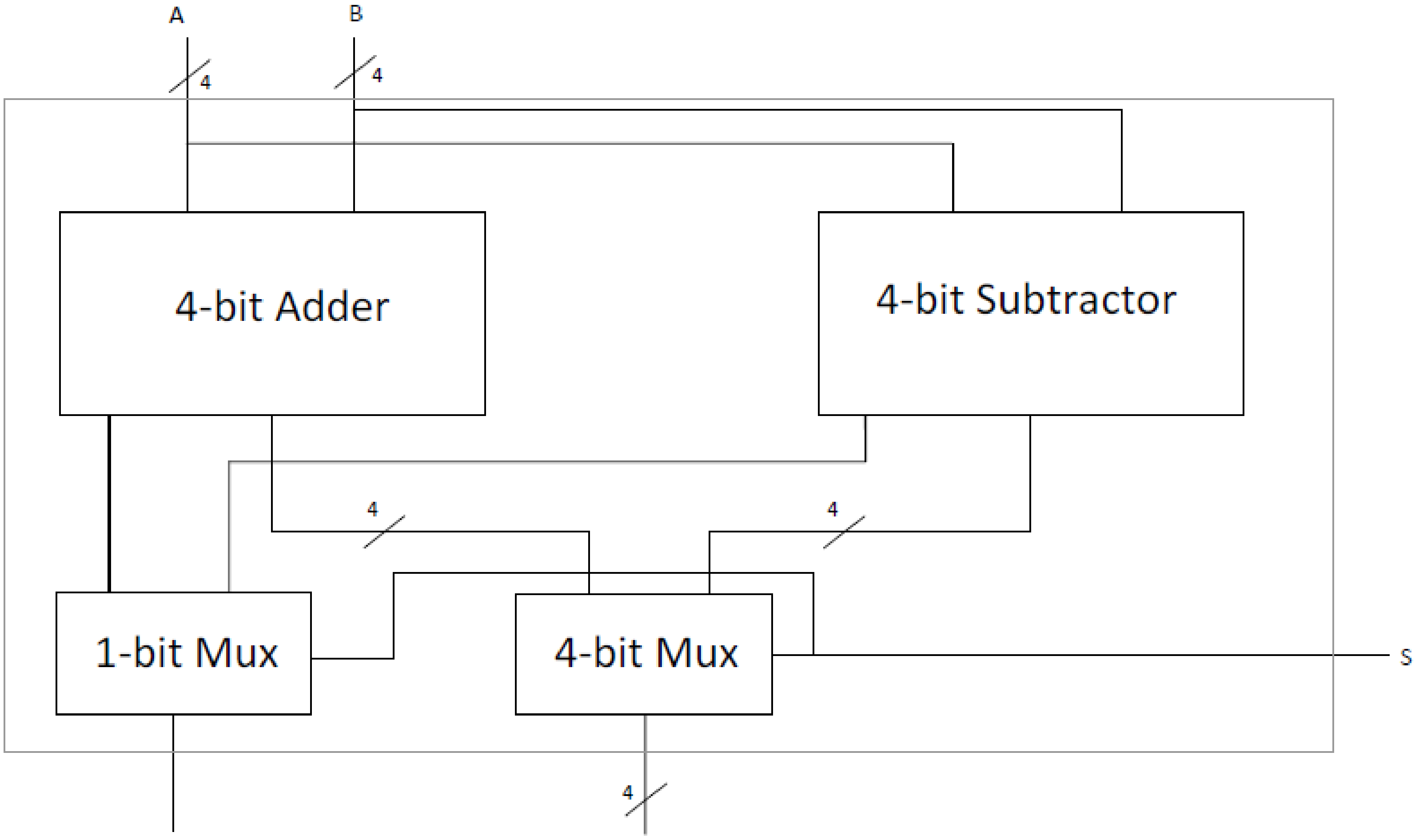
- p.ej., un bit que dice si hay que sumar o hay que restar
- las señales *E* y *S* en los circuitos anteriores

Multiplexer de cuatro entradas (de cuatro bits c/u)

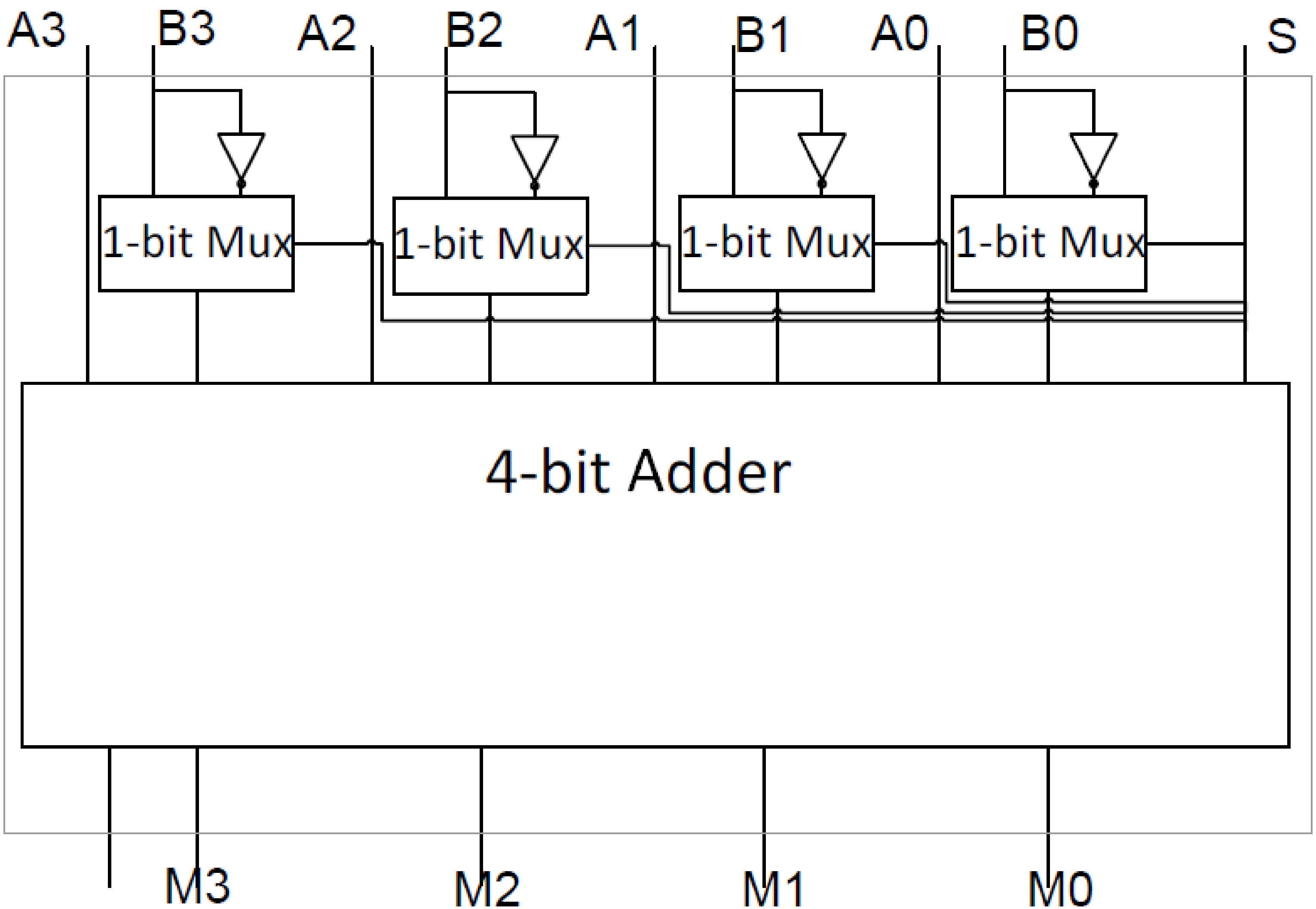
S_0	S_1	*M*
0	0	*A*
0	1	*B*
1	0	*C*
1	1	*D*



Sumador + restador



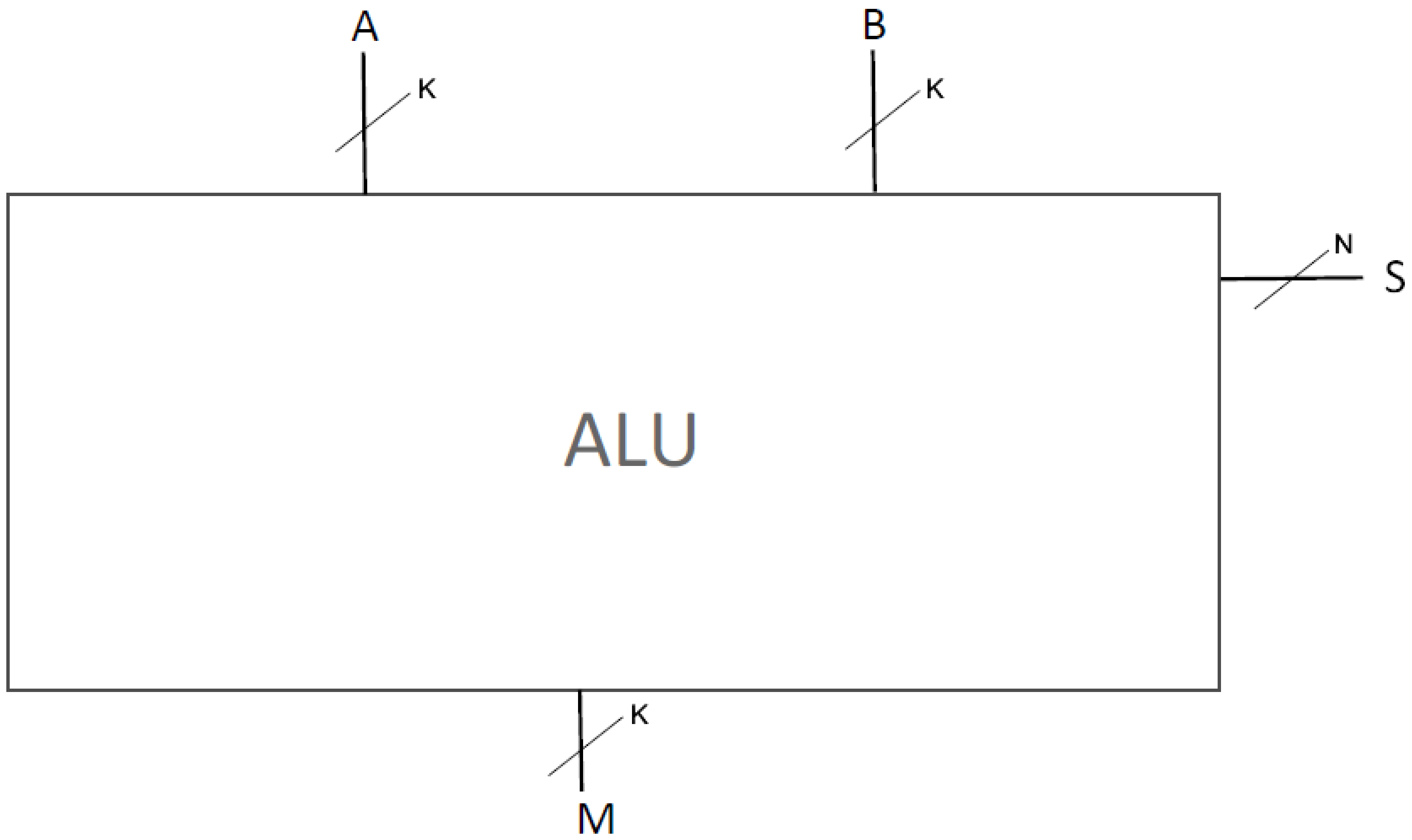
Sumador / restador



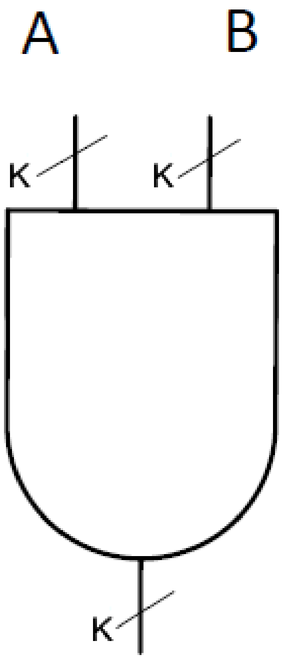
La mayoría (o todos) los computadores tienen un único circuito para realizar operaciones lógicas (p.ej., AND, OR) y aritméticas (p.ej., suma, resta) sobre dos operandos

... → la **unidad lógica aritmética** o **ALU** :

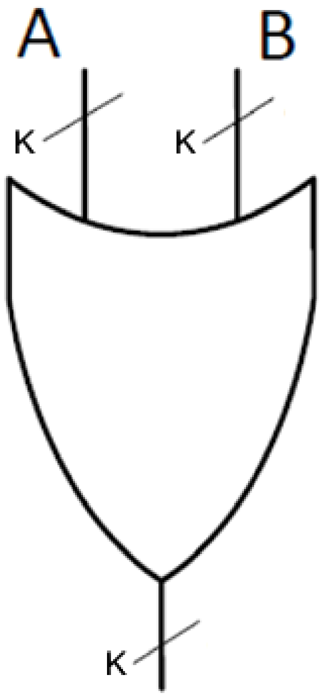
- operandos *A* y *B*, de *k* bits c/u
- resultado *M*, de *k* bits
- bus de control *S*, de *n* bits, permite selecciones entre 2^n operaciones distintas


A
K
B
K
N
S
ALU
K
M

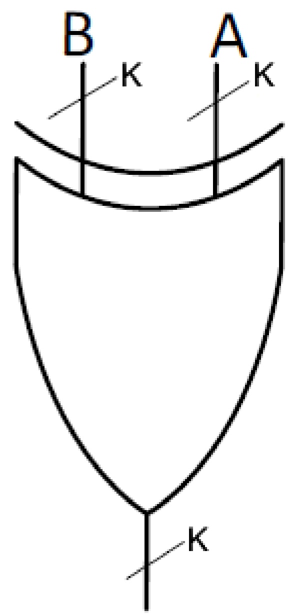
Operaciones *bitwise*



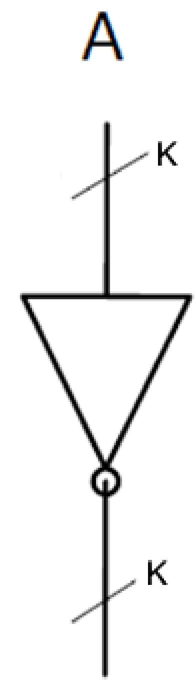
A and B



A or B



A xor B



not(A)

P.ej., una ALU con 8 operaciones

Ejemplos de instrucciones:

- **001**: restar *A* – *B*
- **011**: ejecutar OR entre *A* y *B*
- **110**: multiplicar *A* por 2

S2	S1	S0	M
0	0	0	Suma
0	0	1	Resta
0	1	0	And
0	1	1	Or
1	0	0	Not
1	0	1	Xor
1	1	0	Shift left
1	1	1	Shift right